Generative Adversarial Networks for Sketch and Generation

Harrison Mamin

Abstract

In machine learning, most problems fall into the buckets of classification, regression, or clustering. For this project, I explored a different task: generative modeling. GANs (generative adversarial networks) come in many variants, and I wanted to implement a few different versions in PyTorch and use them to try to generate cat and dog sketches. There are many variants in GAN architectures, but the basic idea is that one model (the Generator) aims to convert some input into an output image that closely matches a chosen distribution of images (the training set). A second model (the Discriminator) serves as a classifier that aims to distinguish between real and generated images. Through a sort of back-and-forth game, the hope is that we eventually reach a point where the Generator is producing images that are indistinguishable from those from the target distribution, and the Discriminator is essentially performing at chance.

I built a DCGAN (deep convolutional GAN) which generates images from random noise, and a CycleGAN which aims to convert between two unpaired sets of images. The industry has not settled on a consensus metric for GAN evaluation, and due to the open-ended nature of this project I opted to rely primarily on the "eye test". By this measure, most results were not convincing enough to fool a human judge, but the best ~10% were difficult to distinguish from real sketches. The models did succeed in learning certain stylistic elements. I experimented with various adjustments such as training the Generator more frequently than the Discriminator, giving the Discriminator a head start before starting to train the Generator, and adopting different activation functions in an aim to avoid vanishing or exploding gradients. A few of these changes showed some promise based on stats I was monitoring – in particular, it often seems helpful to train the Generator more frequently, and this should be done on a batch level rather than an epoch level. Aside from that, my main takeaways were that increasing dataset size and limiting training to a single class at a time (e.g. just sketches of dogs) seemed to help the most. As a future extension of the project, I plan to look into other variations such as Conditional GANs to address this second problem.

Problem

While I applied the models to photo datasets as well, my primary interest here was in seeing if I could "teach a computer to draw", so to speak. My initial hope was to convert from photo to sketch, but ultimately found that the limited number of examples per class in my dataset was an issue here. As a result, I ended up focusing more on generating new sketches that are not based on a particular input (in other words, the Generator operates on random noise rather than photos).

Data

My initial dataset was the Sketchy Database from Georgia Tech. The first portion consists of 12,500 photos evenly split between 125 different categories, including everything from apples and dolphins to pianos and windmills. The second component consists of 75,471 human-drawn sketches (roughly 6 for each corresponding photo). I ran several experiments on this dataset but realized that 100 images per class didn't seem to be enough to produce quality results, so I switched my focus to subsets of the QuickDraw dataset containing sketches from users of a contest-based mobile drawing app. Since classifying dogs vs. cats is a common introductory problem in computer vision, sketching dogs and cats seemed like a reasonable first task for generative modeling. After filtering out the examples that were marked as unrecognized by humans, I was left with 103,030 cat sketches and 143,285 dog sketches. See **Appendix [1]** for sample batches from these data sets.

In the process of testing my models, I also applied them to several other datasets including CIFAR10 (50,000 photos from 10 different classes), CelebA (202,599 photos of celebrity faces), and Da Vinci (142 mechanical sketches from Leonardo DaVinci's notebooks), with varying levels of success.

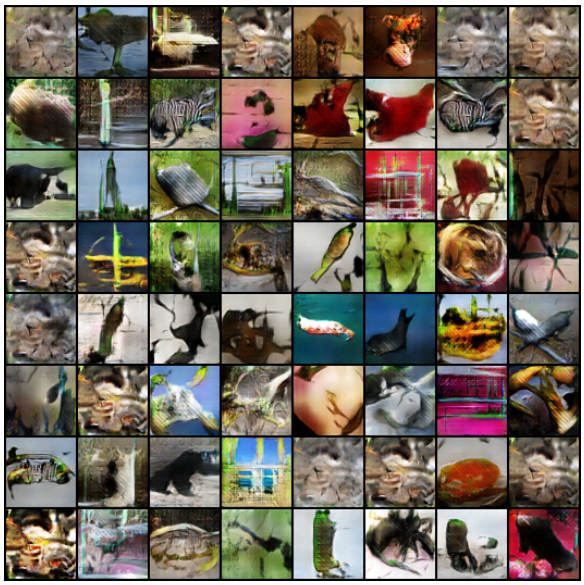
Methods

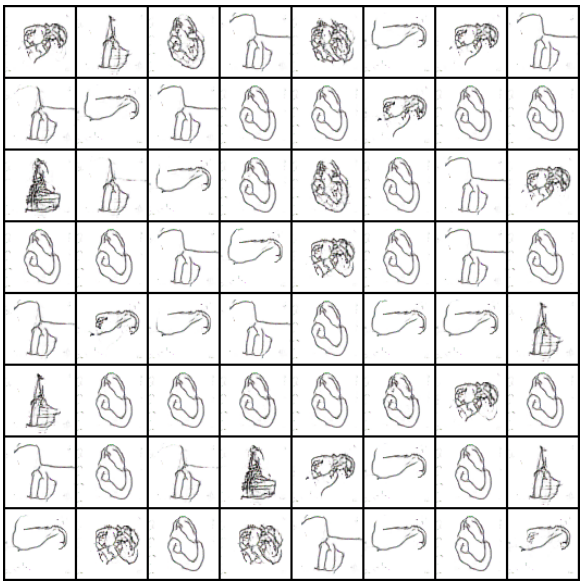
I began by building a PyTorch implementation of a Deep Convolutional GAN (DCGAN). The generator consists of five deconvolutional blocks which gradually upsample the height and width of the input (random noise) into an image. Each block except the last one follows the backward-strided convolution with a batch norm layer (I also add the option to replace batch norm with instance norm). ReLU activations are used after each batch norm layer, while the final deconvolutional block is passed through a tanh function. The discriminator consists of five convolutional blocks which gradually downsample the image height and width as it moves through the network. Another difference from the discriminator is that it uses leaky ReLU activations (recommended leak of .2), and the final activation is a sigmoid to allow for binary classification. At the paper's recommendation, convolutional layers are initialized from a random normal distribution with mean 0 and standard deviation .02. Batch norm initialization uses a mean of 1, which has empirically been shown to work well. My default training function also used other recommended hyperparameters for the learning rate (.0002) and beta_1 for the Adam optimizer (0.5).

I also built a CycleGAN implementation with the initial intent of "translating" photos to sketches using the Sketchy Database. The Cycle Generator architecture is generally described in three stages: the encoder (2 convolutional blocks that increase the number of channels while decreasing height and width), the transformer (2 residual blocks), and the decoder (2 deconvolutional blocks that increase height and width while decreasing the number of channels). I used the same Discriminator as in the DCGAN implementation, though different variations can be used as well. To sum up the basic idea, we have a generator G_xy that converts from a set of images X to set Y, and one G_yx that converts from set Y to set X. One discriminator D_x is trained to distinguish between fake and real x, and D_y distinguishes between fake and real y. G_xy(x) should therefore resemble y and G_yx(G_xy(x)) should resemble X, while G_yx(y) should resemble x and G_xy(G_yx(y)) should resemble y. Ultimately, I did not spend too long tuning these models because by this time I had realized the Sketchy Database did not have enough photos per class to generate strong results.

Experiments and Results

GAN evaluation measures are still a matter of some debate, but on a less rigorous level the "eyeball test" can give some intuition for the quality of results. Below are samples from epoch 300 of training on the photo portion of the Sketchy Database and epoch 60 of the sketch portion. See **Appendix [2]** for samples generated from earlier stages of the training process.

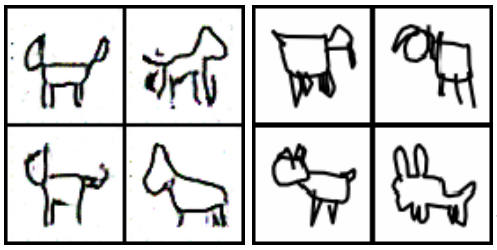




While the vast majority of the output images fail to achieve a level of realism that would fool a human, some learning did occur. At the first epoch, outputs were essentially random noise. By epoch 20, the photo samples showed movement towards bright colors and the outputs became less grainy. As training progressed, image quality continued to sharpen, and at times we saw semi-recognizable features like zebra stripes, flowers, or furry textures. Training on the sketch portion of the Sketchy Database initially looked more promising, with some blurry pencil-like drawings starting to emerge by the end of the first epoch. Over the next few epochs, sharpness improved, and the outputs began to resemble sketches (albeit of unrecognizable targets). However, the Discriminator seemed to be incentivizing the Generator to learn a limited number of shapes as many of the output samples look identical, and the GAN eventually got stuck at a local minimum where the Discriminator predicts every sample as real. In later experiments, I adjusted my training loop to automatically increase the learning rate in these situations to try to escape these sticking points.

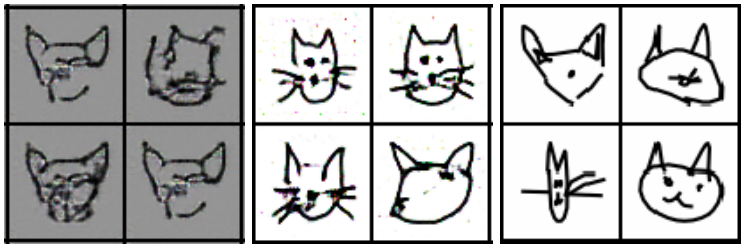
Ultimately, I think this dataset was a challenge due to the relatively low number of examples per class and the fact there is a significant amount of variation between classes. (In comparison, the CelebA Dataset contains only images of faces and results were noticeably better - see **Appendix [3]** – whereas this has everything from airplanes to elephants.) As a positive takeaway, I do think the model learned to mimic the style of the datasets (note the difference in color and line style between the generated photos and sketches); learning the content was the issue. It did produce some interesting visuals, though they're probably closer to modern art than identifiable photos. It's also possible that given more time to train, results could improve – most of my training runs were 1-3 hours at most.

I ran a few more experiments on the Sketchy Database, but due to my concerns about the limited number of samples per class I soon switched my focus to the QuickDraw datasets. This worked noticeably better. The dog samples were, to my eye, a bit weaker, but this was true of the real samples as well in my opinion – people seemed to have an easier time drawing identifiable cats than dogs. By comparing our fake samples to some of the weaker (but still recognized) real samples below, we can see that the model managed to produce some outputs that could plausibly pass as real.



Left: Dog samples from epoch 7 of Trial 4, using JReLU activations, lr_d of .0001, and gd_ratio of 2.*****
Right: Real dog samples from QuickDraw dataset.

*****_JReLU_: ReLU variant described in more detail below.
_lr_d_: Discriminator's learning rate (lr default was .0002 for both discriminator and generator).
_gd_ratio_: # of times Generator is trained for each time Discriminator is trained, described in more detail below.

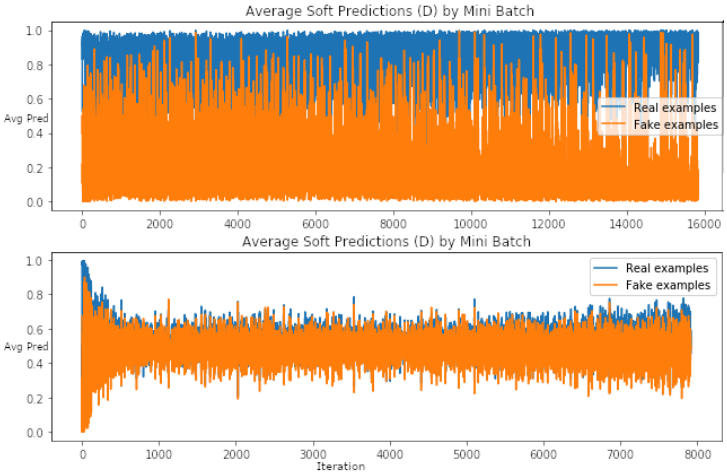


Left: Cat samples from epoch 5 of Trial 1, using mostly default hyperparameters (grey background due to normalizing with stats from dataset – changed in later trials).
Middle: Cat samples from epoch 10 of Trial 4, using JReLU activations, lr_d of .0001, and gd_ratio of 2.
Right: Real cat samples from QuickDraw dataset.

One variation I tried was replacing the standard ReLU or leaky ReLU activations with a modified version proposed by FastAI's Jeremy Howard. This activation uses a leak of 0.1, subtracts 0.4 from the output, and sets an upper threshold of 6.0. The potential benefit here is that it may do a better job at maintaining a mean of zero and variance of one for the activations as they move through the network, diminishing the risk of exploding or vanishing gradients.

Another variation I tried was using a pre-trained model for the Discriminator. One reason GANs can be challenging to train is that each model's training relies on other model(s). Convergence often takes several epochs even for vanilla neural networks, and as a result GANs often end up relying on extremely poor teachers for the earlier parts of training. My hope was that using transfer learning to give the Discriminator a head start might allow us to skip this frustrating stalemate between two largely untrained models. To do this, I used a ResNet18 with a sigmoid activation at the end. GANs require us to keep multiple models in memory, so I thought it was best to avoid the larger ResNet34, DenseNets, or Inception architectures at least initially. So far, I've had limited success with this approach, but have not yet ruled out its potential – most of the recommended hyperparameters and training methods assume an untrained Discriminator, so I would likely need to experiment with many different variations to reach a more firm conclusion.

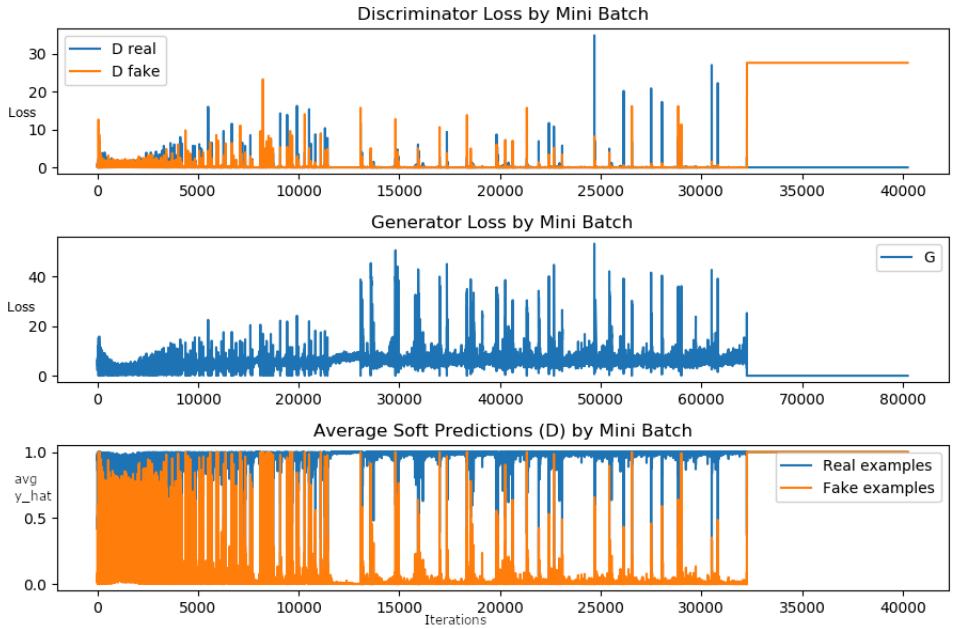
Similarly, I tried training only the Discriminator for the first few batches to try to ensure that it had something useful to teach the Generator. I did not notice much of a meaningful difference in training. A more impactful change was varying the ratio of Generator to Discriminator training (gd_ratio). A ratio of 2 seemed to work well, meaning that the Generator was trained on every mini batch and the Discriminator alternated between being trained and not trained. This resulted in the Discriminator's mean soft predictions more often converging to 0.5 for both real and fake examples, as desired.



The top figure shows the Discriminator's mean soft predictions when being trained on every mini batch of the CelebA dataset. The bottom figure resulted from training the Discriminator on every other mini batch. CelebA's more consistent results made it useful for testing.

Conclusion

On the Sketchy Database, the large number of classes and relatively small amounts of data per class made it difficult for the simple DCGAN models to learn the class contents. However, they did do a good job mimicking the style of each dataset. When training on single classes from the larger QuickDraw datasets, results were much better. Empirically, training the discriminator every other mini batch seemed to work well on these datasets, and it resulted in the discriminator's soft predictions approaching 0.5 for both real and fake data, as desired. Some tuning is required, and generated image quality didn't always reflect this supposed improvement. Using a smaller learning rate for the discriminator was also effective at times. Some of the best results when generating cat and dog sketches combined both of these changes, training the discriminator half as often as the generator and with half the learning rate. They also used Jeremy Howard's ReLU variant in place of the standard leaky ReLU. An additional tweak that proved helpful at times was a gate in the training loop that occasionally checks if the losses have stopped changing. GANs can have a tendency to get stuck, and by increasing the learning rate in these situations we hope to jump out of these local minima. This did not always work, however, and I generally found GANs to be quite finnicky to train.



GANs getting "stuck" was a common problem, as seen around iteration 32,000 in the results above.

Going forward, there are still countless experiments I could run with my existing models, as well as many different GAN variants that I could try out. In particular, I want to focus more on models that take class labels into account so I can train on many categories simultaneously. In addition, I'd like to learn more about interpreting quantitative metrics like inception score or MS-SSIM. I primarily stuck with visual evaluation methods due to the experimental nature of this project and the fact that most GAN metrics have faced significant criticisms, but that is an area to consider moving forward.

Code

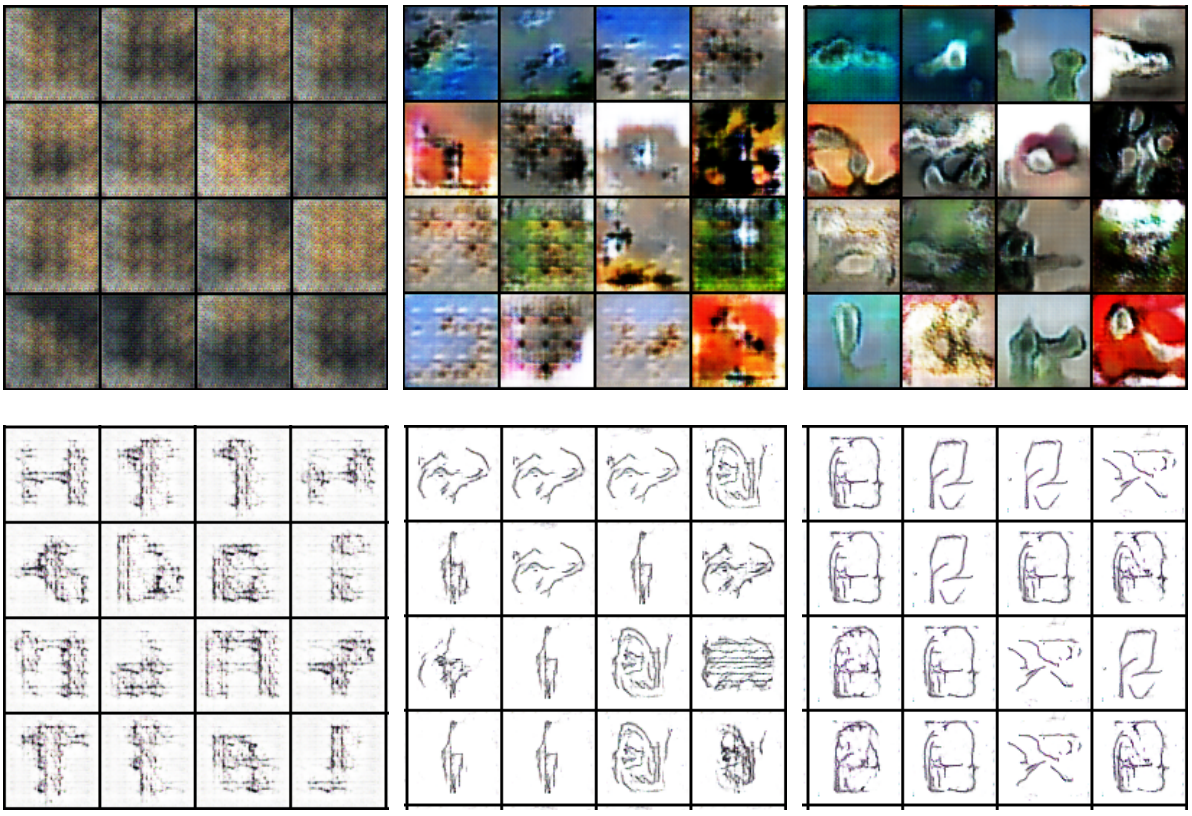
https://github.com/hdmamin/GAN-architectures

Note: Jupyter notebooks were used primarily for the development phase, while the finalized code was saved in .py files. Most experiments were run from the command line and samples were saved in image directories not uploaded to Github. As the only team member, all project responsibilities were handled by Harrison Mamin.

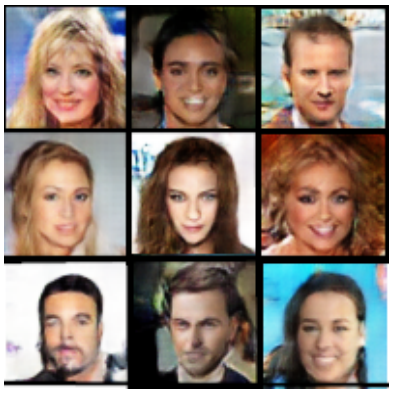
Appendix

[1] Sample batches from various datasets. From left to right and top to bottom: Sketchy Database photos, Sketchy Database sketches, QuickDraw cats, QuickDraw dogs, CelebA, CIFAR10.

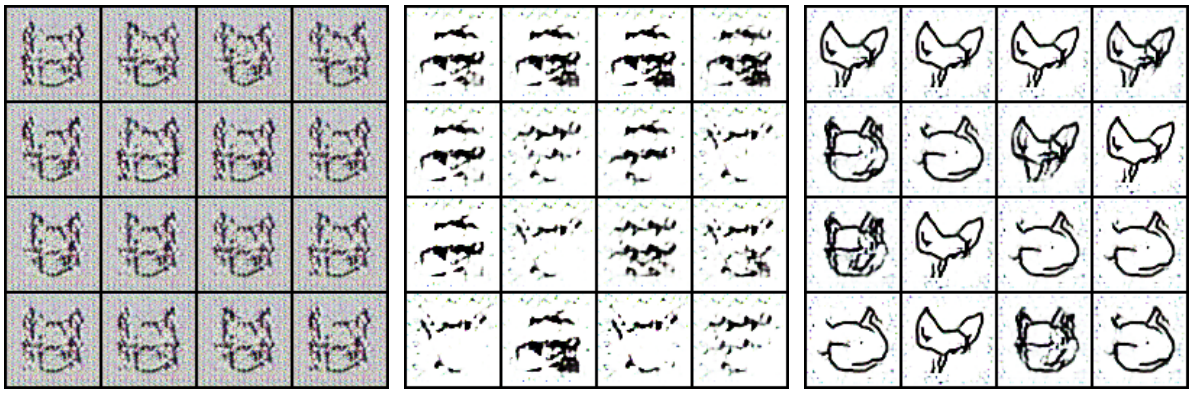
[2] Generated samples from epochs 0, 10, and 20 from training on the Sketchy Database. All settings and hyperparameters used default values on this run.
Top row: Photo portion
Bottom row: Sketch portion



[3] Samples generated from the CelebA dataset. These were all created during the first 4 epochs.



[4] Samples from epoch 0, 5, and 10 of Trial 2 on the QuickDraw cat dataset. This is a good example of the unpredictability of GAN results, and also shows the models' tendency to repeat shapes.



Sources

https://arxiv.org/pdf/1703.10593.pdf
https://distill.pub/2019/gan-open-problems
https://hardikbansal.github.io/CycleGANBlog